Parallel Computing Final-Term: *Mean Shift*

Giovanni Bindi

Università degli Studi di Firenze

24 February 2021

Introduction: *Mean Shift*

Mean shift is a popular non-parametric clustering technique developed during the 1970s [1]. It aims to locate the *modes* of a density function. Given a set of observations $\mathcal{S} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ with $\mathbf{x} \in \mathbb{R}^D$ and a kernel function $K_h : \mathbb{R}^D \to \mathbb{R}$ it operates a kernel density estimation:

$$f(\mathbf{x}) = \frac{1}{Nh^D} \sum_{i=1}^{N} K_h\Big(\frac{\mathbf{x} - \mathbf{x}_i}{h}\Big) \tag{1}$$

where $h \in \mathbb{R}$ is called the window radius (or width).

Idea

The modes of the density function are located at the zeros of the gradient $\nabla f(\mathbf{x}) \to$ derive an iterative algorithm for clustering [2].

Algorithm

Define the the weighted mean of the density, determined by K_h as

$$m(\mathbf{x}) = \frac{\sum_{\mathbf{x}_i \in N(\mathbf{x})} K_h(\mathbf{x} - \mathbf{x}_i)\mathbf{x}_i}{\sum_{\mathbf{x}_i \in N(\mathbf{x})} K_h(\mathbf{x} - \mathbf{x}_i)} \tag{2}$$

where $N(\mathbf{x})$ is a neighborhood of $\mathbf{x}$. Typically a gaussian kernel is used

$$K_\sigma(\mathbf{x} - \mathbf{x}_i) = e^{\frac{\|\mathbf{x} - \mathbf{x}_i\|}{2\sigma^2}} \tag{3}$$

For $t = 1, \dots, I$ iterations, being $\mathcal{S}^{(1)} = \mathcal{S} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$:

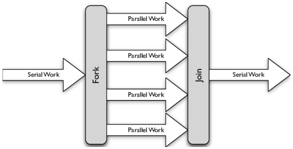
1. **Calculate** $m(\mathbf{x}_i^{(t)})$, for each $\mathbf{x}_i^{(t)} \in \mathcal{S}^{(t)}$
2. **Update** $\mathbf{x}_i^{(t+1)} = m(\mathbf{x}_i^{(t)})$, for each $\mathbf{x}_i^{(t)} \in \mathcal{S}^{(t)}$

At the end reduce $\mathcal{S}^{(I)}$ to a set of clusters with centers $\{\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$.

Pseudo-code: Sequential Version

```
Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$
1: for $t = 1, \dots, I$ do
2:     for $i = 1, \dots, N$ do
3:         $\boldsymbol{\tau} = \mathbf{0}$
4:         $\eta = 0$
5:         for $j = 1, \dots, N$ do
6:             if $\|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|_2^2 \leq R$ then
7:                 $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})\mathbf{x}_j^{(t)}$
8:                 $\eta = \eta + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})$
9:             end if
10:        end for
11:        $\mathbf{x}_i^{(t+1)} = \boldsymbol{\tau}/\eta$
12:    end for
13: end for
14: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow \texttt{reduce\_centroids}(\mathcal{S}^{(I)}, \delta)$
15: return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$
```

OpenMP [3]

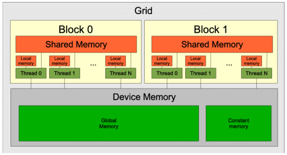


- Take advantage of **fork-join** via OpenMP.
- Only **few** directives needed for this task:
  - `#pragma omp parallel for`
  - `#pragma omp critical`
- Benefit from both **shared** and **private** memory.
- Exploit both workload sharing mechanism:
  - **Static** (set by the user).
  - **Dynamic** (set by OpenMP).

OpenMP: Pseudo-code

```
Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Optional: number of threads $T \in \mathbb{N}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$
1: for $t = 1, \dots, I$ do
2:     #pragma omp parallel for (dynamic or $T$)
3:     for $i = 1, \dots, N$ do
4:         $\boldsymbol{\tau} = \mathbf{0}$
5:         $\eta = 0$
6:         for $j = 1, \dots, N$ do
7:             if $\|\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)}\|_2^2 \leq R$ then
8:                 $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})\mathbf{x}_j^{(t)}$
9:                 $\eta = \eta + K_\sigma(\mathbf{x}_i^{(t)} - \mathbf{x}_j^{(t)})$
10:            end if
11:        end for
12:        #pragma omp critical
13:        $\mathbf{x}_i^{(t+1)} = \boldsymbol{\tau}/\eta$
14:    end for
15: end for
16: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ reduce_centroids($\mathcal{S}^{(I)}, \delta$)
17: return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$
```

CUDA [4]



- Take advantage of **SIMT** model.
- Data stored as a $N \times D$ array $\rightarrow$ 1-dimensional `blockDim`.
- User decides number of threads $T \rightarrow \lceil N/T \rceil$ blocks.
- Exploit both **global** and **shared** memory accesses.

CUDA: Pseudo-code

```
Input: $\mathcal{S} \subset \mathbb{R}^D$. $I \in \mathbb{N}$. $\sigma, R, \delta \in \mathbb{R}$.
Input: Threads per block: $T \in \mathbb{N}$.
Output: Centroids $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$ for some $k \in \mathbb{N}$
1: $B = \lceil N/T \rceil$
2: for $t = 1, \dots, I$ do
3:     mean_shift«B,T»($\mathcal{S}^{(t)}, \sigma, R$)
4:     cudaDeviceSynchronize()
5: end for
6: $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\} \leftarrow$ reduce_centroids($\mathcal{S}^{(I)}, \delta$)
7: return $\{\boldsymbol{\mu}_1, \dots \boldsymbol{\mu}_k\}$
```

CUDA: *Naive*

```
Input: $\mathcal{S}^{(t)} \subset \mathbb{R}^D$. $\sigma, R \in \mathbb{R}$.
1: tid = (blockIdx.x $\times$ blockDim.x) + threadIdx.x
2: if tid $<$ N then
3:     $\boldsymbol{\tau} = \mathbf{0}$
4:     $\eta = 0$
5:     for $i = 1, \ldots, N$ do
6:         if $\|\mathbf{x}^{(t)}_{\mathtt{tid}} - \mathbf{x}^{(t)}_i\|^2_2 \leq R$ then
7:             $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}^{(t)}_{\mathtt{tid}} - \mathbf{x}^{(t)}_i)\mathbf{x}^{(t)}_i$
8:             $\eta = \eta + K_\sigma(\mathbf{x}^{(t)}_{\mathtt{tid}} - \mathbf{x}^{(t)}_i)$
9:         end if
10:    end for
11:    $\mathbf{x}^{(t+1)}_{\mathtt{tid}} = \boldsymbol{\tau}/\eta$
12: end if
```

CUDA: *Shared Memory*

```
Input: $\mathcal{S}^{(t)} \subset \mathbb{R}^D$. $\sigma, R, \in \mathbb{R}$.
  $\omega$ = threadIdx.x
  tid = (blockIdx.x $\times$ blockDim.x) + $\omega$
  $\mathcal{M} \leftarrow$ shared array[$T \times D$]
  $\mathcal{V} \leftarrow$ shared array[$T$]
  $\boldsymbol{\tau} = \mathbf{0}, \eta = 0$
  for $\tau = 1, \dots, \lceil N/T \rceil$ do
      $\gamma = \tau \times T + \omega$
      if $\gamma < N$ then
          $\mathcal{M}_\omega^{(\tau)} = \mathbf{x}_\gamma^{(t)}$ else $\mathcal{M}_\omega^{(\tau)} = \mathbf{0}$
          $\mathcal{V}_\omega^{(\tau)} = 1$ else $\mathcal{V}_\omega^{(\tau)} = 0$
      end if
      __syncthreads()
      for $i = 1, \dots, T$ do
          $\mathbf{x}_i^{(t)} \leftarrow$ load_data_from_sm($\mathcal{M}^{(\tau)}$)
          $v_i^{(t)} \leftarrow$ load_multiplier_from_sm($\mathcal{V}^{(\tau)}$)
          if $\|\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)}\|_2^2 \leq R$ then
              $\boldsymbol{\tau} = \boldsymbol{\tau} + K_\sigma(\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)})\mathbf{x}_i^{(t)}$
              $\eta = \eta + K_\sigma(\mathbf{x}_{\mathtt{tid}}^{(t)} - \mathbf{x}_i^{(t)})v_i^{(t)}$
          end if
      end for
      __syncthreads()
  end for
  if tid < N then
      $\mathbf{x}_{\mathtt{tid}}^{(t+1)} = \boldsymbol{\tau}/\eta$
  end if
```

Experiments

- CPU: 4C / 8T (Intel i7-860).
- GPU: NVIDIA GTX 980.
- No ϵ-stop $\rightarrow I = 50$ iterations.
- Ten runs for each experiment:

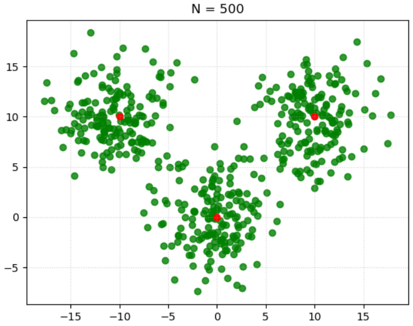
$$\bar{t}_A = \frac{1}{10} \sum_{i=1}^{10} t_A^{(i)} \qquad (4)$$

- Speedup of B over A:

$$S = \bar{t}_A / \bar{t}_B \qquad (5)$$

- Variance:

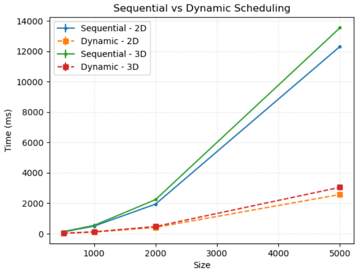
$$\sigma^2 \approx S^2 \left[\frac{\sigma_A^2}{\bar{t}_A^2} + \frac{\sigma_B^2}{\bar{t}_B^2} \right] \qquad (6)$$



- Four dataset sizes $N = \{500, 1000, 2000, 50000\}$.
- Two dimensionalities $D = \{2, 3\}$.
- Three centroids.

General Behaviour

Algorithm time complexity: $O(IN^2)$



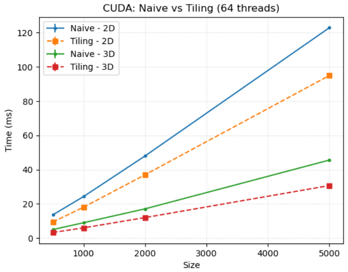
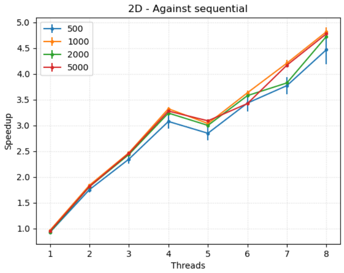


Figure 1: Execution time (*ms*): OpenMP (*left*) (dynamic scheduling) - CUDA (*right*) (with $T = 64$)

OpenMP: Against Sequential



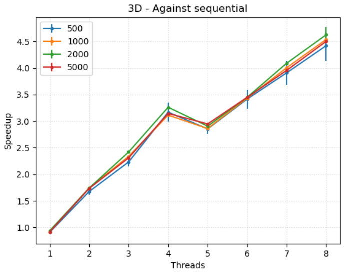
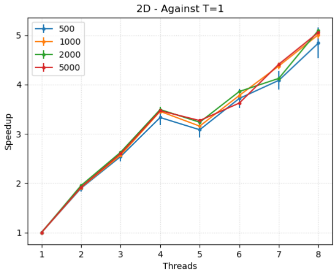


Figure 2: Speedup: Parallel vs Sequential for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 4.82$ for $N = 1000$, $D = 2$ and $T = 8$.
- **Minimum** speedup: $S^{min} = 0.91$ for $N = 500, 1000, 5000$, $D = 3$ and $T = 1$.

OpenMP: Against $T = 1$



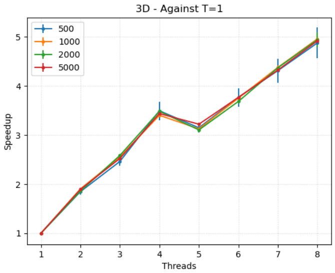


Figure 3: Speedup: Parallel vs $T = 1$ for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 5.10$ for $N = 2000$, $D = 2$ and $T = 8$.
- **Minimum** speedup: $S^{min} = 1, \quad \forall N, D$ and $T = 1$.

CUDA: Naive Against Sequential

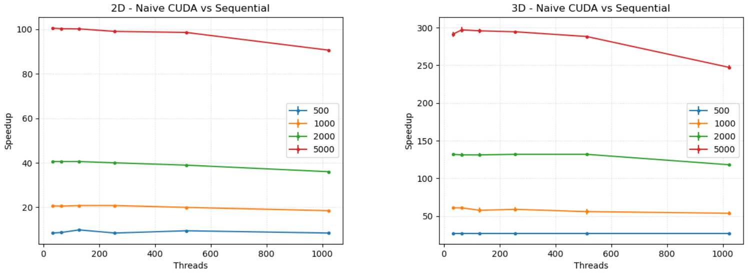
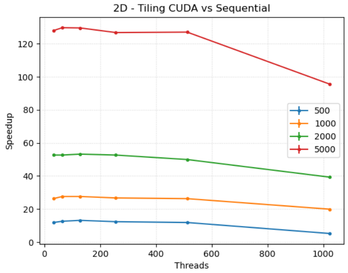


Figure 4: Speedup: Naive vs Sequential for 2D (*left*) and 3D (*right*) datasets

- **Maximum** speedup: $S^{max} = 297.1$ for $N = 5000$, $D = 3$ and $T = 64$.
- **Minimum** speedup: $S^{min} = 8.46$ for $N = 500$, $D = 2$ and $T = 32, 256, 1024$.

CUDA: Tiling Against Sequential



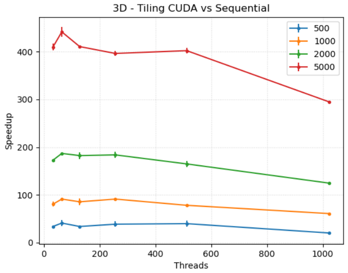


Figure 5: Speedup: Tiling vs Sequential for $2D$ (*left*) and $3D$ (*right*) datasets

- **Maximum** speedup: $S^{max} = 441.32$ for $N = 5000$, $D = 3$ and $T = 64$.
- **Minimum** speedup: $S^{min} = 11.85$ for $N = 500$, $D = 2$ and $T = 32, 512$.

Conclusions

Three implementations of *mean shift*: one sequential and two parallel.

Pros:

- Parallelization methods: OpenMP & CUDA.
- Header-only C++17 libraries, with simple test cases.
- OpenMP: simplest implementation.
- CUDA: obtained a considerable speedup.

Cons:

- OpenMP: speedup not comparable wrt CUDA.
- CUDA: implementation is more difficult.

Code

https://github.com/w00zie/mean_shift

References I

[1] K. Fukunaga and L. Hostetler, "The estimation of the gradient of a density function, with applications in pattern recognition," *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975. DOI: 10.1109/TIT.1975.1055330.

[2] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, 2002. DOI: 10.1109/34.1000236.

[3] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming,", vol. 5, no. 1, pp. 46–55, Jan. 1998, ISSN: 1070-9924. DOI: 10.1109/99.660313. [Online]. Available: https://doi.org/10.1109/99.660313.

[4] NVIDIA, P. Vingelmann, and F. H. Fitzek, *Cuda, release: 11*, 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

Code

Centroids reduction:

```cpp
template<typename T, const size_t D>
double calc_distance(const vec<T, D>& p, const vec<T, D>& q) {
    double sum = 0.0;
    for (size_t i = 0; i < D; ++i)
        sum += ((p[i] - q[i]) * (p[i] - q[i]));
    return sum;
}

template <typename T, const size_t  D>
bool is_centroid(std::vector<vec<T, D>>& curr_centroids, const vec<T, D>& point, const double eps_clust) {
    return std::none_of(curr_centroids.begin(),
                        curr_centroids.end(),
                        [&](auto& c) {return calc_distance(c, point) <= eps_clust;});
}

template <typename T, const size_t N, const size_t D>
std::vector<vec<T, D>> reduce_to_centroids(mat<T, N, D>& data, const float min_distance) {
    std::vector<vec<T, D>> centroids = {data[0]};
    for (const auto& p : data) {
        if (is_centroid(centroids, p, min_distance))
            centroids.emplace_back(p);
    }
    return centroids;
}
```

Code - Sequential

```cpp
template <typename T, const size_t N, const size_t D>
std::vector<vec<T, D>> cluster_points(mat<T, N, D>& data,
                                      const size_t niter,
                                      const float bandwidth,
                                      const float radius,
                                      const float min_distance) {
    const float double_sqr_bdw = 2 * bandwidth * bandwidth;
    mat<T, N, D> new_data;
    for (size_t i = 0; i < niter; ++i) {
        for (size_t p = 0; p < N; ++p) {
            vec<T, D> new_position {};
            float sum_weights = 0.;
            for (size_t q = 0; q < N; ++q) {
                double dist = calc_distance(data[p], data[q]);
                if (dist <= radius) {
                    float gaussian = std::exp(- dist / double_sqr_bdw);
                    new_position = new_position + data[q] * gaussian;
                    sum_weights += gaussian;
                }
            }
            new_data[p] = new_position / sum_weights;
        }
        data = new_data;
    }
    return reduce_to_centroids(data, min_distance);
}
```

Code - OpenMP

```cpp
template <typename T, const size_t N, const size_t D>
std::vector<vec<T, D>> cluster_points(mat<T, N, D>& data,
                                      const size_t niter,
                                      const float bandwidth,
                                      const float radius,
                                      const float min_distance) {
    const float double_sqr_bdw = 2 * bandwidth * bandwidth;
    mat<T, N, D> new_data;
    for (size_t i = 0; i < niter; ++i) {
        #pragma omp parallel for default(none) \
        shared(data, niter, bandwidth, radius, double_sqr_bdw, new_data) \
        schedule(dynamic)
        for (size_t p = 0; p < N; ++p) {
            vec<T, D> new_position {};
            float sum_weights = 0.;
            for (size_t q = 0; q < N; ++q) {
                double dist = calc_distance(data[p], data[q]);
                if (dist <= radius) {
                    float gaussian = std::exp(- dist / double_sqr_bdw);
                    new_position = new_position + data[q] * gaussian;
                    sum_weights += gaussian;
                }
            }
            #pragma omp critical
            new_data[p] = new_position / sum_weights;
        }
        data = new_data;
    }
    return reduce_to_centroids(data, min_distance);
}
```

Code - CUDA Naive

```
__global__ void mean_shift(float *data, float *data_next) {
    size_t tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    if (tid < N) {
        size_t row = tid * D;
        float new_position[D] = {0.};
        float tot_weight = 0.;
        for (size_t i = 0; i < N; ++i) {
            size_t row_n = i * D;
            float sq_dist = 0.;
            for (size_t j = 0; j < D; ++j) {
                sq_dist += (data[row + j] - data[row_n + j]) * (data[row + j] - data[row_n + j]);
            }
            if (sq_dist <= RADIUS) {
                float weight = expf(-sq_dist / DBL_SIGMA_SQ);
                for (size_t j = 0; j < D; ++j) {
                    new_position[j] += weight * data[row_n + j];
                }
                tot_weight += weight;
            }
        }
        for (size_t j = 0; j < D; ++j) {
            data_next[row + j] = new_position[j] / tot_weight;
        }
    }
    return;
}
```

Code - CUDA Shared Memory

```
__global__ void mean_shift_tiling(const float* data, float* data_next) {
    // Shared memory allocation
    __shared__ float local_data[TILE_WIDTH * D];
    __shared__ float valid_data[TILE_WIDTH];
    // A few convenient variables
    int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
    int row = tid * D;
    int local_row = threadIdx.x * D;
    float new_position[D] = {0.};
    float tot_weight = 0.;
    // Load data in shared memory
    for (int t = 0; t < BLOCKS; ++t) {
        int tid_in_tile = t * TILE_WIDTH + threadIdx.x;
        if (tid_in_tile < N) {
            int row_in_tile = tid_in_tile * D;
            for (int j = 0; j < D; ++j) {
                local_data[local_row + j] = data[row_in_tile + j];
            }
            valid_data[threadIdx.x] = 1;
        }
        else {
            for (int j = 0; j < D; ++j) {
                local_data[local_row + j] = 0;
                valid_data[threadIdx.x] = 0;
            }
        }
        __syncthreads();
        ...
```

Code - CUDA Shared Memory 2

```
...
for (int i = 0; i < TILE_WIDTH; ++i) {
            int local_row_tile = i * D;
            float valid_radius = RADIUS * valid_data[i];
            float sq_dist = 0.;
            for (int j = 0; j < D; ++j) {
                sq_dist += (data[row + j] - local_data[local_row_tile + j]) *
                    (data[row + j] - local_data[local_row_tile + j]);
            }
            if (sq_dist <= valid_radius) {
                float weight = expf(-sq_dist / DBL_SIGMA_SQ);
                for (int j = 0; j < D; ++j) {
                    new_position[j] += (weight * local_data[local_row_tile + j]);
                }
                tot_weight += (weight * valid_data[i]);
            }
        }
        __syncthreads();
    }
    if (tid < N) {
        for (int j = 0; j < D; ++j) {
            data_next[row + j] = new_position[j] / tot_weight;
        }
    }
    return;
}
```